Computer Science & Software Engineering

CITS4407/CITS2003 OPEN SOURCE TOOLS AND SCRIPTING

Assignment 2 2022

Submission deadline: 11:59pm, Monday 23 May 2022.
Value: 30% of CITS2003/CITS4407.
To be done individually.

This assignment will involve creating a program based on Bash, Awk, Sed and other Unix tools, as appropriate. It makes sense to decompose the problem and use the most appropriate Unix tool for each part. The top-level script has been given a name. Please make sure you use the specified name as that is the name that we will use to test your program. **You need to package all of these into a single submission consisting of a zip or a tar file, and submit the zip/tar file *cssubmit*. Please use the CITS4407 submission box as the the submission boxes for the two classes are still aliased. No other method of submission is allowed.** (Please do not submit any test files, as these will be provided.)

You are expected to have read and understood the University's guidelines on academic conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort. Plagiarism detection, and other systems for detecting potential malpractice, will therefore be used. Besides, if what you submit is not your own work then you will have learnt little and will therefore, likely, fail the final exam.

You must submit your project by the submission deadline listed above. Following UWA policy, a late penalty of 5% will be deducted for each day (or part day), after the deadline, that the assignment is submitted. However, in order to facilitate marking of the assignments in a timely manner, no submissions will be allowed after 7 days following the deadline.

Quantifying and Comparing Writing Style

Overview

UWA, like every university around the country (probably around the planet) is very worried about ghost-written submissions for assignments. This is also known as contract cheating. (UWA's statement is similar.) Indeed, provision of contract cheating services is illegal in Australia. Whatever you call it, ghost-writing is about getting someone else to do your work, but submitting it as if it was your work. The incidence is believed to be low, but it's clearly not a good thing. In this case we are concerned with essays.

Rather than looking directly at the content of texts, as one does when looking for suspected plagiarism, one way of tackling this problem is to look for stylistic similarities. That is, one looks for similarities in the ways particular authors use language, rather than similarities in the actual words on the page, on the assumption that an author will use a similar style for similar sorts of content, fiction, non-fiction, etc. In order to do this computationally, we need to define metrics, quantifying (and then comparing) linguistic features that contribute to an author's style.

Overview of what is to be done

What you will do for this Assignment is write a program, consisting of Shell, Awk and Sed scripts, and calls to other Unix tools, as appropriate, that reads in either one or two text files containing text to be analysed. The top level Shell program should be called `style_cmp`. If one file is specified, a *profile* should be created and printed out **to standard output**. (More will be said about profiles below.) If two files are specified, a profile should be created for each, but rather than printing the profiles, your program should compare the profiles using a metric, which will be described below, and a score should be reported **on standard output**. The score reflects the distance between the two works in terms of their style; low scores, down to 0, imply that the same author is likely responsible for both works, while large scores imply different authors.

What is a word?

For this program a word is either a simple word, a word-pair, a contraction, or a possessive, where a simple word is a sequence of one or more alphabetic characters followed by a space, or any punctuation other than hyphen or apostrophe, or the end of the file. A word-pair is two or more simple words separated by hyphens with no spaces around them. Word-pair is a word-pair.

A possessive is a simple word followed by an apostrophe (i.e. in effect single quote), which in turn is immediately followed by "s", e.g. "mother's". A contraction is a simple word followed by an apostrophe, followed by other alphabetics, e.g. "won't", "shouldn't"

What is a sentence

A sentence is is a sequence of one or more words followed by either a full-stop, question-mark or exclamation mark.

Creating a profile

- Your program is to count the number of occcurences of certain words and certain pieces of punctuation. Specifically, the list of words to be counted is:
  `"also", "although", "and", "as", "because", "before", "but", "for", "if", "nor", "of", "or", "since", "that","though", "until", "when", "whenever", "whereas", "which", "while", "yet"`

  If you are wondering why that particular set of words is being used, they are conjunctions, which can indicate more complex sentences without relying on the content of the text.

- In addition to the set of conjunctions listed above, your program needs to count words, as defined above: simple or compound words, possessives or contractions; each count as 1 word.
- Beyond their inclusion in the total word count (above), your program should separately count contractions (but not possessives) and compound words.
- Your program should count certain pieces of punctuation: comma and semicolon.
- Finally, your program needs to count the number of sentences

What is a Profile

A profile is a text file containing lines of the form:
style-marker count
where the style-markers are the various items that you need to count. The list of markers is:

- Count of each of the words listed above. Instances should be counted regardless of capitalisation, reported in lower-case.
- word (All the words. Each compound-word, possessive and contraction counts as 1 word.)
- compound_word
- contraction
- comma
- semi_colon
- sentence

To make it easier to inspect profiles, the profiles should be shorted in alphabetic order.

Comparing profiles

As mentioned earlier, if the names of two text-files are submitted, a profile should be computed for each file. Then, the values for most of the style-markers need to normalised, because longer text will have larger counts of the different style-markers. Normalised profiles will differ from the computed profiles in the following ways:

- The counts for each of the conjunctions, comma, semi_colon, compound_word and contraction be normalised by dividing by the overall word count **and multiplying by 1,000**, to give count per 1,000 words.
- The count of words will be ignored.
- The count of sentences will be replaced by the number of words divided by the number of sentences, to give words per sentence.

The two normalised profiles will then be compared using the standard Euclidian Distance metric:

$$dist = \sqrt{\sum_i \left(profile1_i - profile2_i\right)^2}$$

That is, the square root of the sum of the squares of the differences between corresponding style-metrics from a pair of profiles.

A couple small examples

Profile creation

The first example, is a profile of a sample taken from Adventures of Huckleberry Finn (Mark Twain). Here is the corresponding profile. A second sample, taken from Kangaroo (D. H. Lawrence), results in this profile.

Text (i.e. normalised profile) comparison

Here are three slightly longer samples: Kangaroo Children of the Bush (Henry Lawson) and Three Elephant Power (Andrew Barton "Banjo" Patterson. The corresponding profiles are: Kangaroo sample profile, Children of the Bush sample profile, and Three Elephant Power sample profile

While not visible to the user, each of the profiles must first be normalised (as described above). For example, the profile of Children of the Bush sample is normlised to this profile

However, what the user will see when comparing texts is the output of the Euclidian Distance comparion of the two normalised profiles. For example, comparing the Lawrence and

Lawson samples I get: this. The only relevant part is the final line, which reports the distance. What I have also chosen to report are the squares of the *normalised* style-metric differences, which suggests that the big contributors to the final distance are use of comma and "and" (at least for those to samples). The Patterson versus Lawrence sample comparison indicates they have a similar level of difference, but for different items: compound-word and contraction. However the Patterson versus Lawson sample comparison suggests they are much more different, where again it appears to be highly dependent on compound-word, contraction and "and". Remember: these are measures of dissimilarity, not similarity.

By the way, so long as your program reports the distance, the other information can be ignored, if you wish.

To give you something longer to work with, here is the profile for the entire text of Hucklebery_Finn.txt.

Tips, Tricks, Traps, Suggestions

While the discussion in the sections above has mostly been a specification of what your program needs to do, the discussion in this section is really in the nature of suggestions, which you can choose to follow, or not, as you wish. The suggestions are in no particular order.

Double hyphen

As mentioned above, when the text was being captured by OCR, the symbol that printers call an em-dash — (i.e. a long dash), has been rendered as "--" (double hyphen). Given that we are interested in word-pairs, it may be advisable to get rid of double hyphen early, e.g. by conversion to space character.

Sample texts

As I have done above, don't try tackling complete texts. Rather, grab short sections where there are syntactically interesting things going on, and test your code with these. Being short, you can work out by yourself what you'd expect to see, given the definitions above of word and sentence, etc. For example, `sample1.txt` contains "St. Lawrence". English speakers know that St. is a contraction of Saint (or Street), but that is well beyond this unit. From our point of view, it looks like a full-stop after St.

However, when you are ready to test your program on complete texts rather than samples, here is another set of texts from Project Gutenberg.

How to think about/structure your program?

Given that there are two (related) functions to be implemented, it seems to me that a good way to think about it is as two sub-scripts, one implementing profile creation, and the other implementing profile comparison, with a Shell script on top that does the front-of-house stuff.

Profile creation

I think this is the harder of the two tasks. The way you may care to do this is first to transform the text into a sequence of things you want to keep: words and the bits of punctuation we are interested in, one per line. You then use a Sed script to convert these words, punctuation, etc, into a stream of tokens, e.g. "word" presents all words, "comma" represents ",", etc. This script is, arguably, the trickiest part of the project. Spend some time on it and do a lot of testing with diffent text selections. Once you have the stream of tokens, a fairly simple Awk script can be created to compute the profile. Remember to sort the profile.

Profile comparison

Given the structured, and sorted, profiles, this should be fair straight forward. My only suggestion here is that don't try to do the normalisation step and the comparison step in the one Awk script. Instead, create a normalise script to normalise each of the two profiles. Then combine the normalised profiles and feed that into the final Euclidian Distance script.

Marking Rubric

Your program will be marked out of 30.

- 66% of the marks (20/20) will be awarded based on how well your program completes a number of tests, reflecting normal use of the program, and also how the program handles various error states, such as the input file not being present. Other than things that you were asked to assume, you need to think creatively about the inputs your program may face.
- 33% (10/30) will be style/maintainability (5/10) — the code is clear to read — and efficiency (5/10) — your program is well constructed and runs efficiently. For style, think about use of comments, sensible variable names, your name at the top of the program. (Please look at your lecture notes, where this is discussed.)

  Much of this has been discussed in classes, but includes comments, meaningful variable names for meaningful variables (i.e. not throw away variables such as loop variables), and sensible anti-bugging. It also includes making sure your program removes any temporary files that were created along the way.

Style Rubric
For the style/maintainability mark, the rubric is:

$0 \leq x < 1$	Gibberish, impossible to understand
$1 \leq x < 2$	Style is really poor, but can see where the train of thought may be heading
$2 \leq x < 3$	Style is acceptable with some lapses
$3 \leq x < 4$	Style is good or very good, with small lapses
$4 \leq x < 5$	Excellent style, really easy to read and follow

- Your program will be traversing text files of various sizes (possibily including large corpora) so try to minimise the number of times your program reads the same file.

  Efficiency Rubric

$0 \leq x < 1$	Code too incomplete to judge efficiency, or wrong problem tackled
$1 \leq x < 2$	Very poor efficiency, many file traversals
$2 \leq x < 3$	Acceptable efficiency, one or more lapses slowing the code
$3 \leq x < 4$	Good efficiency, small lapses
$4 \leq x < 5$	Excellent efficiency, has no problem with large files

- Automated testing is being used so that all submitted programs are being tested the same way. Sometimes it happens that there is one mistake in the program that means that no tests are passed. If the marker is able to spot the cause and fix it readily, then they are allowed to do that and your - now fixed - program will score whatever it scores from the tests, minus 2 marks, because other students will not have had the benefit of marker intervention. Still, that's way better than getting zero, right? (On the other hand, if the bug is too hard to fix, the marker needs to move on to other submissions.)

Department of Computer Science & Software Engineering
The University of Western Australia
Last modified: 18 May 2022
Modified By: Michael Wise

UWA